Vrije Universiteit Brussel

Faculteit Wetenschappen

Effort Estimation for Changing Requirements

Patrick Steyaert, Wim Codenie, Wilfried Verachtert

Techreport vub-prog-tr-96-11

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Effort Estimation for Changing Requirements

Patrick Steyaert
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2
B1050 Brussel, BELGIUM
e-mail: prsteyae@vnet3.vub.ac.be

Wim Codenie, Wilfried Verachtert
OO Partners BVBA,
Otto De Mentockplein 19
B1853 Strombeek-Bever, BELGIUM
e-mail: {wilfried, wim}@oopartners.com

Abstract: *Complex software systems have to cope with a steady stream of changing requirements. However, current software engineering techniques are ill equipped for assessing the impact of changes, either within a single system or across reused assets and the systems in which they are reused. This is a critical inhibitor for (amongst others) effort estimation in the object-oriented development process where reuse and incremental development play a crucial role. We have been confronted with this problem in deriving a commercial domain specific framework from a custom built application (in the area of broadcast planning). In this case the problem boils down to efficient effort estimation of customisations of the framework early in the development process. In the context of a research project we are investigating reuse contracts as a possible solution.*

Background

This section describes the problems we encountered at OO Partners in the process of turning a custom developed planning application for a TV-station (VTM) into a complete domain specific framework that has been successfully installed at several TV-stations in Europe.

The project goal was to create an application to support the planning of TV-programs. More specific, the application should integrate *all* the aspects of planning, ranging from seasonal planning to daily planning, as well as video tape management, program information management and downloading information to specific transmission hardware. In other words, the application should be accessible by different kinds of users basically manipulating the same data but from a different point of view. Another requirement was that the application should be as close as possible to the planning processes used at VTM. At that time most of the planning was still done with the aid of pencil and paper. The application should improve the quality of the work —e.g. automate the work flow, reduce administrative overhead— and increase productivity, but not influence the basic procedures that were already used for planning.

The project started early 1993 with a young team of 6 software engineers. Our first prototype was immediately taken into production. Since the initial release, several updates were done, mainly to add functionality. The final version was taken into production end 1994.

Even before the final version of PSI was installed at VTM, another Belgian and several European TV-stations showed interest in the product. Marketing research has revealed that few software solutions are available on the market for scheduling and planning TV-programs and that most of the existing off-the-shelf software is based on older technologies. They lack sophisticated graphical interfaces, don't reflect the dynamic work flow and are not easy to adapt to client-specific needs within limited time and budget. Especially the latter shortcoming is considered a serious problem in the rapidly evolving TV-market. This observation motivated us to build a domain specific framework that is easily customisable.

Major Requirement Changes

Our first major modification of PSI was done at VTM for the opening of their second channel. Because the board of directors wanted to keep their ideas for the second channel secret for their direct competitors, only very few people were informed. The official announcement was made only one month before the start of the new channel. Even the planning and IT departments didn't know of the plans in advance. Needless to say that there was some panic among the users. In one month the existing single-channel planning application had to be transformed into a multi-channel planning application[1]. This had not only implications on the planning strategies used —e.g. ensuring that a film is planned only on one of the channels— but also on some of the domain components —e.g. transmission dates had to be associated with a channel.

In the mean time, a customised version of PSI has been successfully installed at two other broadcast stations, referred to as case 1 and case 2. Both versions are derived from the original application. In case 1 only twenty percent of the classes had to be adapted. These adaptations mostly consisted of adding specific features. In the other case, the existing work flow procedures had to be revised, but the underlying domain objects could be reused. These two extra cases allowed iteration which was very important for OO Partners to gradually transform this customised application into a framework. The reason for this incremental approach was to limit the upfront investments in the framework.

Our Experience with Building the Framework

The increasing interest in PSI is for our company a big challenge on the field of software engineering. How to keep the PSI framework manageable —i.e. maintain, support and add new features to it— and in the same time respond to the demands for adapting PSI towards specific needs?

The need to make PSI adaptable requires the definition of a clear architecture expressed in terms that are understandable to experts in the domain. It is the materialisation of our years of experience in the field and includes information about the work flow, the kinds of actions planners can perform and what they expect from other users of the application. In our case the architecture is expressed as a framework consisting of abstract classes, contracts between classes, and numerous hot spots that serve as hooks for future installations of the framework.

It is often suggested that the ultimate goal of this kind of framework development is to build - through a small number of iterations - a software architecture that can be turned into a customised application by simply filling in the hot spots. It should be possible to implement the requirements that differ from case to case without altering the framework itself. In our experience this is a false delusion. Even after the initial iterations modifications to the framework still occur, albeit less frequently. Although these latter changes occur less frequently, they are the hardest to assess when building customised applications. Most often such changes have a large impact on the rest of the system and are often incompatible with previous customisations. This makes the management of the consistency between the different customisation and the maintenance of the framework itself extremely difficult.

Because of the difficulties we are currently very interested in methods for assessing the impact of changes:

---

[1] This is a good example of what it means for the IT department to be able to evolve in parallel with the business.

- Estimating what parts of the software can be reused and what parts must be modified (e.g. what methods can be inherited, what methods must be added/overridden)

- How modifications propagate through other parts of the system or reusers of the modified parts

In a joint research project we have been investigating reuse contracts as a means to facilitate the propagation of changes to reusable assets and indicating where and how to test and how to adjust these applications. We are currently interested in investigating how reuse contracts can be used for effort estimation for changing requirements.

Reuse Contracts

In [Steyaert&al96] we propose reuse contracts as a means to codify the management of change in an (adaptable) software system. Reuse contracts record the protocol between builders and customisers of an adaptable system and offer guidelines for deriving more transient (customised) versions of the system in some problem domain. Similar to real world contracts that can be amended, extended, or customised, reuse contracts are subject to typical reuse operations such as refinement, extension and concretisation. The inverse operations: coarsening, cancellation and abstraction intuitively correspond to the (partial) breaching of a contract.

Reuse contracts and their operations are used to document how a transient version has been derived from the more persistent parts of the system. This documentation can be used by tools to asses the impact of changes made to the system, to forecast when and which problems might occur and to give directions on where and how to test the derived transient version of a system. For example, when extending the adaptable system with new functionality, collisions with already made extensions in more transient versions must be checked; or, when cancelling functionality in the system, it needs to be checked whether no transient versions rely on such functionality. To be able to check this, more information is needed on how a transient version relies on the design decisions made in the adaptable system. Reuse contracts provide exactly this information.

Because the best-known technique available today for structuring and adapting object-oriented software is undoubtedly the use of abstract classes with inheritance as the reuse mechanism, in [Steyaert&al96] we focused on the problem of reuse of class-hierarchies as a more tangible case to express the ideas behind reuse contracts. In that context, reuse contracts and their operations describe the protocol between managers and users of (abstract) class libraries. Reuse contracts of abstract classes provide an explicit representation of the design decisions behind an abstract class, including information such as: which methods can be sent to the class, which methods are invoked by what other methods, which methods are abstract or concrete, relationships with other classes, ... Only information relevant to the design is included. For example, auxiliary or implementation-specific methods are not mentioned in a reuse contract.

Reuse contracts can be manipulated by means of reuse operations. Refinement refines the design of some methods, extension adds new methods, concretisation makes abstract methods concrete. These reuse operations not only allow documenting the adaptations made to a class, but a careful investigation of their interactions also allows to predict and manage the effect of these adaptations.

Effort Estimation: An Example

Consider the example of a Collection hierarchy. A class Set defines a method add and a method addAll to add a collection of elements simultaneously. The specification of our set class is provided in an OMT class diagram.

```
Class Set
    method      add(Element) = 0
    method addAll(aSet:Set) =
      begin
        for e in aSet do
            self.add(e)
      end
end
```

This Set class is part of our application. Due to requirement changes we are asked to modify this set class to count the number of elements in a set. In order to estimate the effort we need to estimate how much of the existing methods need to be modified. In worst case we need to rewrite both add and addAll methods. When either add or addAll depends in its implementation on the other method, we might be able to reuse one method. The OMT diagram does not provide sufficient information for this analysis as it does not state the dependencies between the method implementations. Because of the simplicity of the example code inspection works fine here. In practice inspecting the code for effort estimation is undesirable. This kind of analysis should be feasible on the design level. An intermediate level of description is needed. Reuse contracts for classes can provide this information. In a reuse contract each method can have a specialisation clause (in italics in the example) that documents, at the design level, how it depends on the other methods. The reuse contract is an interface description to which the implementation must comply.

```
reuse contract Set
    abstract
        add(Element)
    concrete
        addAll(Set) {add(Element)}
end
```

This reuse contract gives more precise information for effort estimation than the OMT diagram. The question is whether this reuse contract gives us sufficient information to decide what can be reused and what must be newly developed without error-prone code inspection. Reuse contracts will indicate which methods rely on which other methods, by enumerating the names of methods that are invoked through self sends. The reuse contract of the example does not state that the add method is invoked for each element of the argument set. Although reuse contracts provide only syntactic information it is our experience that in practice this is enough to estimate what methods can be inherited and what methods must be overridden. The art is in finding the right balance between descriptions that are easily understood and expressed and descriptions that capture enough of the semantics of possible adaptations.

For simplicity we have restricted the reuse contracts we propose here to include only documentation on the internal dependencies among a class's methods. The dependencies among the methods of one class and the methods of its acquaintances are at least as important. Just as our current reuse contracts are based on specialisation interfaces, reuse contracts are being developed based on descriptions of interclass relationships.

Based on the information provided by the reuse contract the following effort estimate can be made.

```
--- effort estimate
# subclassing Set
# concretisation of add(Element)
```

Reuse contracts can also be used to asses the impact of changes or updates of reusable assets. Let's say that we are going to make an optimised version of Sets. In the optimised version addAll stores the added elements directly rather than invoking the add method. This leads to inconsistent behaviour in CountingSet when it decides to upgrade to this optimised set as not all additions will be counted. Using the terminology of [Kiczales&Lamping92] we say that addAll and add have become *inconsistent methods*. Again, although in this simple example this can be derived from the code, in practice it should be possible to detect these problems without code inspection. The major obstacle for locating problems such as inconsistent methods is that the different conceptual ways to reuse an (abstract) class are all performed by the same operation, i.e. inheritance. More information about the intentions of inheritor is needed. This kind of information is precisely provided by the reuse operations on reuse contracts. In the example the reuse contracts of CountableSet and the OptimisedSet document how they were derived from Set.

```
reuse contract CountableSet concretises Set
    concrete
        add(Element)
end

reuse contract OptimizedSet coarsens Set
    concrete
        addAll(Element) {-add(Element})
end
```

The fact that add and addAll have become inconsistent can be derived directly from the reuse contracts: CountableSet concretises a method that has been removed from the specialisation clause while changing from the old parent class to the new parent class (in italics above). In this example only two reuse operations are used: concretisation and coarsening. In [Steyaert&al96] a complete set of reuse operators is given together with a set of rules that allow automatic detection of conflicts based on the interaction of reuse operations.

Again an effort estimate can be given on the basis of the reuse contracts. An inconsistent method conflict has a fixed cost in most cases as it can be resolved by a standard rule.

```
--- effort estimate for replacing Set with OptimisedSet
#Resolving the inconsistent method conflict between add and
addAll in class CountingSet by:
   1) copying the addAll method from OptimisedSet to
      CountingSet
   2) merging the code for counting elements (from add in Set)
      in this addAll method
```

Conclusion

In the current state-of-the-art impact of changes can only be estimated by informal reasoning, mostly at the code level. The result is that subtle conflicts with important consequences are often only detected during the testing phase. This is obviously not a good basis for effort estimation. When adopted, reuse contracts can make effort estimation of changes feasible without code inspection and by formal reasoning and automated tool support.

In situations where reuse and incremental development are an issue, reuse contracts do not necessarily create an overhead. We see reuse contracts as an embracing methodology for managing the different aspects (testing, effort estimation, ...) of change and reuse.

While we only discussed how reuse contracts can improve the predictability of the impact of change we are confident that they can actually be used as a basis for more formal estimation metrics for changes. The reason is that reuse contracts allow the

detection of conflicts of different categories (e.g. name conflicts, inconsistent methods, method capture, partial implementation,... in the case of inheritance) and allow a more fine grained estimation of what needs to be changed and what not. In our experience in using reuse contracts the different categories can be associated different metrics. Obviously, for those conflicts that can only be resolved by adding new methods or classes other metrics must be used.

5 References

[Garlan&Shaw95] D.Garlan, M.Shaw: An Introduction to the Field of Software Architecture, In V.Ambriola and G.Tortora, eds., Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing Company, 1995.

[Goldberg&Rubin95] A.Goldberg, K.S.Rubin: *Succeeding with Objects Decision Frameworks for Project Management*, ISBN 0-201-62878-3, Addison-Wesley Publishing Company, Inc., 1995.

[Kiczales&Lamping92] G. Kiczales, J. Lamping: *Issues in the Design and Specification of Class Libraries*, Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 435-451, ACM Press 1992.

[Lieberherr96a] K.J.Lieberherr: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, ISBN 0-534-94602-X, PWS Publishing Company, Boston, 1996.

[Lieberherr96b] K.J.Lieberherr: *From Transience to Persistence in Object Oriented Programming: Architecture and Patterns*, Position Statement for the working group on object-oriented programming within the ACM Workshop on Strategic Directions in Computing Research, to be held at MIT, June 1996 and for the ECOOP'96 Adaptability workshop.

[Pancake95] C.M.PanCake: *Object Roundtable, The Promise and the Cost of Object Technology: A Five-Year Forecast*, In Communications of the ACM, October 1995, Vol 38(10), pp. 32-49, ACM Press, 1995.

[Steyaert&al.96] P.Steyaert, C.Lucas, K.Mens, T.D'Hondt: *Reuse Contracts: Managing the Evolution of Reusable Assets,* To appear in Proceedings of OOPSLA'96 Conference on Object Oriented Programming, Systems, Languages and Applications, ACM Press 1996.

[Yourdon94] E.Yourdon: *Object-Oriented System Design: An Integrated Approach*; Yourdon Press Computing Systems, Prentice Hall, 1994.